

Programare orientata-obiect

C++

Curs 1

Virginia Niculescu

Structura cursului

- Elemente de baza ale limbajului C++.
- Tipuri de date derivate si utilizator si alocare dinamica in C++.Programare modulara in C++.
- Metoda programarii orientate-obiect in C++.
- Supraincarcarea operatorilor. Membrii statici. Functii Friend
- Clase: containere si iteratori
- Relatia de mostenire
- Clase abstracte si Polimorfism
- Sabloane (Template). Biblioteca STL
- Operatii de intrare/iesire.
- Tratarea exceptiilor.
- Sabloane de proiectare
- Dezvoltare bazata pe testare. Instrumente de testare

Paradigma POO (1)

Necesitatea:

- Spatiul problemei \leftrightarrow Spatiul solutiei
- Modelul problemei \leftrightarrow Modelul solutiei

POO ofera instrumente eficiente pentru reprezentare

POO permite rezolvarea problemei in termenii problemei,
nu in termeni specifici modelului calculatorului pe care
se executa solutia.

Obiect = reprezentarea unui element din spatiul problemei
(def. generala) in spatiul solutiei

- Orice obiect are o **stare**
- Orice obiect are o **interfata**

Paradigma POO(2)

■ **Concepte:**

1. *Clasa* - implementare a unui TAD
2. *Obiect* = instanta a unei clase
3. *Metoda* = implementarea unei operatii din interfata TAD
(*mesaj* = apel al metodei → interactiune intre obiecte)

■ **Caracteristici:**

1. *Incapsulare* (date + operatii)
ascunderea informatiei ⇒ protectia informatiei ⇒ consistenta datelor
2. *Mostenire* ⇒ *reutilizarea codului*
3. *Polimorfism* = capacitatea unei entitati de a reactiona diferit in functie de starea sa.

Avantaje ale POO

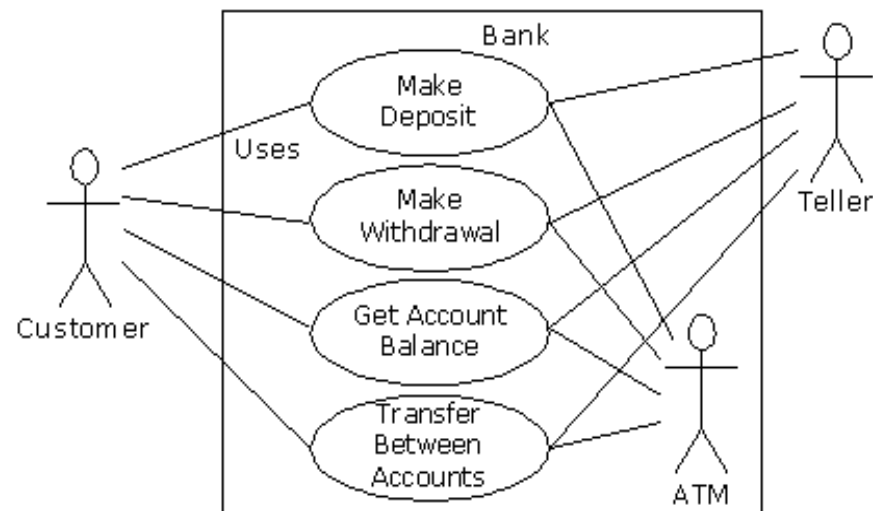
- **Abstractizare** (de nivel înalt)
 - Un obiect poate fi utilizat fără a se cunoaște reprezentarea sa internă
- **Modularitate & Modificabilitate**
 - Codul sursă corespunzător unui obiect poate fi scris și întreținut independent de codul sursă corespunzător altor obiecte.
 - O clasă poate fi ușor schimbată fără a se modifica alte clase.
 - Biblioteci de clase
- **Lizibilitate**
 - Ușor de înțeles -> de depanat, întreținut
- **Reutilizabilitate**

!!! Doar dacă programele sunt construite respectând principiile POO!!!

Analiza si Proiectare

- “Cine va folosi sistemul?” (pentru a descoperi actorii)
- “Ce pot face acesti actori cu sistemul?”
- “Cum poate sistemul reactiona daca altcineva face acestea?”
(pentru a descoperi variatiile)
- “Ce probleme pot apare in sistem?”
(pentru a descoperi exceptiile)

1. Descoperirea obiectelor
2. Asamblarea obiectelor
3. Constructia sistemului
4. Extensia sistemului
5. Reutilizarea obiectelor



Analiza si Proiectare Orientata pe Obiecte

- **Scop:** creare de sisteme informatice bine proiectate, robuste, usor de intretinut folosind tehnologii OO.

Analiza si Proiectare OO vs. Analiza si Proiectare Orientata pe Functii

- **Descompunere** (divide&conquer) strategia primara folosita pentru a stapani complexitatea sistemelor informatice.
 - Analiza si proiectare structurata:
 - descompunere bazata pe functii sau procese
 - Analiza si proiectare OO :
 - descompunere bazata pe obiects sau concepte

Programare Imperativa <-> Programare Orientata-Obiect

- **Descompunere programului in subprograme**(proceduri, functii)=> steps

- **Action-oriented** —

- Sarcini

- Identificarea **actiunilor**/functiilor de baza
- Implementarea actiunilor *to do tasks* ca subprograme (proceduri/functii/)
- Gruparea **subprogramelor** in programe/module/biblioteci
- dezvolt. unui **a sistem complet** pentru rezolvarea problemei

- **Top-down**

- **Identificarea obiectelor**
- **Identificarea interactiunii dintre obiecte in vederea rezolvarii problemei**

- Focus pe entitatile din domeniul problemei Sarcini:

- Determinarea **objects** necesare/implicate
- Determinarea modului in care ele pot lucra impreuna pentru rezolvarea problemei
- Crearea tipurilor corespunzatoare **classes** =>
 - **date membru**
 - **functii membru**
- Instantierea tipurilor
- Interactiune intre obiecte — transmitere de mesaje.

- **Bottom-up**

Limbaje puternic tipizate versus slab tipizate

- Tip de date
- Variabile – instantieri
- Conversii implicite
- ***Dynamic type-checking***
 - ex. -o variabila poate stoca fie o valoare numerica fie o valoare booleana a variable might store either a number or the Boolean value "false".(poate fi considerat "weakly typed") *Python*
- ***Static type-checking***
 - specificarea tuturor tipurilor folosite de program
- In C se impune declararea oricarei variabile inainte de folosire
- Alte limbaje folosesc *type inference* – ex. *Haskell*
- *intermediar- C#*

- ***Limbajul C++***

Structura unui program

- Directive de procesare, #
- Declarații de date globale,
- Declarații de funcții, [Antete de funcții (prototipuri) ;]
- Funcția principală(main)
↓
[Descrierea funcțiilor (implementări)]

- Înainte de compilare, un program este **precompilat**, de către un preprocesor, care permite includerea unor fișier sursă, definirea și apelul unor macrouri, precum și o compilare condiționată.

- **Includerea** unui fișier sursă (*.h sau *.cpp) se realizează prin directiva **include** astfel:

include "specificator_fișier" // pentru fișiere utilizator

- sau

include <specificator_fișier> // pentru fișiere standard

- Exemplu:

#include <stdio.h>; // **Standard Input Output Header**

#include <iostream.h>; // **Input & Output Stream**

#include <conio.h>; // **Console Input, Console Output**

Funcții

- O funcție este formată dintr-un antet și un bloc (corp). Ea poate fi apelată dacă a fost definită în întregime sau doar antetul său.

- **Antetul** unei funcții are următorul format:

<tip> <Nume> (<lista_parametri_formali>)

unde:

- Tip este tipul valorilor funcției (codomeniul);
- Nume este un identificator (literă urmată eventual de alte litere sau cifre);
- Listă_parametri_formali conține parametrii formali separați prin ‘,’ (**virgulă**).

- Exemplu:

int Min (int a, int b)

{ if (a<b) return a; else return b; }

- declarație versus definiție!

Funcții

- **Corpul** unei funcții are următoarea structură:

```
{  
  Declarații  
  Instrucțiuni  
}
```

- Exemple:

```
int Cmmdc(int a, int b) // Cmmdc(a,b)  
{  
    if (b==0) return a;  
    else return Cmmdc(b,a % b); // Cmmdc(b,a Mod b);  
}  
int cmmdc(int a, int b) // cmmdc(a,b)  
{ int rest;  
  do { rest=a%b;  
      a=b;  
      b=rest; }  
  while (rest!=0); // rest ≠ 0; sau while (rest) ;  
  return a;  
}
```

Elemente de limbaj de baza

- **Alfabetul** limbajului C este format din litere mari și mici, cifre și caractere speciale (`\n=CrLf`, `\t=Tab`).
- **Identificatorii** sunt formați din literă_ urmată eventual de litere_ sau cifre (caracterul ‘_’ poate fi utilizat pe post de literă).
- Există **cuvinte cheie** care pot fi utilizate doar în contextul definit (de exemplu `case`, `float`, `int`, `long`, `return`, `short`, `static`, `structure`, `switch`, `union`, `unsigned`, `void`).
- **Tipurile predefinite**
 - `int`
 - `short` ,
 - `long` ,
 - `unsigned` ,
 - `float` ,
 - `double` ,
 - `char` (cod ASCII).
- `sizeof()`

Expresii

- O **expresie** este formată din **operanzi**, **operatori** și **paranteze** pentru prioritate, și are o **valoare** și un **tip**.
- **Asocierea** operatorilor se face de la stânga la dreapta, cu excepția operatorilor unari și de atribuire, care se asociază de la dreapta la stânga.
- **Operanzii** pot fi: constante, constante simbolice, variabile simple sau structurate (tablouri, structuri, sau elemente ale acestora), funcții sau apeluri de funcții.

Constante

- **Constantele numerice** pot fi zecimale (123, 123**L**ong, 111**l**ong), octale (**0**77), hexa (**0x**abba, **0XBABA**), sau flotante (2.71828, 6.023**e**23, 6.023**E**23).
- **Constantele** de tip **caracter** pot fi afișabile ('A', '0', '"') sau funcționale ('\b'=Backspace, '\r'=Return, '\n'=Newline, '\''=Apostrof, '\\'=Backslash, '\v'=Verticaltab, '\f'=Salt de pagină, '\0'=Null).
- **Constantele** de tip **șir de caractere** se scriu între ghilimele ("Mesaj").

Declaratii si definitii

Declaratie =

asociere intre un nume si un tip

<type> <identifier>

unde

- **<type>** tip
- **<identifier>** numele variabilei

Exemple:

```
int a, int b; //abreviere: int a,b;  
int length (char[ ]); // function declaration  
class Student;
```

Definition =

o declaratie +

o constructie a unei entitati (alocare spatiu de memorie) cu numele declarat

Exemple:

```
- int a; (declaration + definition)  
  
- int length(char a[] ) {...// definition  
    }  
  
class Student {...};
```

Declararea variabile

- **Declararea variabilelor simple :**

Tip Listă_identificatori_de_variabile;

- Exemple:

`int i, j; float x,y; char c;`

- **Declararea unui tablou :**

Tip Nume_Tablou [d1] [d 2] ... [di] ... [dn]; // indicele k_i : $0 \leq k_i < d_i$

- Exemple:

`float x[100]; x[0]=1; ... x[99]=100; // x este pointer la primul element`

`int a[2][2]; a[0][0]=1; a[0][1]=2; // a conține adresa tabloului`

`a[1][0]=3; a[1][1]=4; 05.03.06 4`

Tipuri de date

Tipuri:

- i. de baza (simple, predefinite)
- ii. derivate (pointer, vector, reference)
- iii. definite de utilizator (struct, class)

Basic Types – tipuri numerice

- *char* -
- *short, int, long* – nr. intregi
- *float, double* – nr. reale

$\text{sizeof(char)} \leq \text{sizeof(short)} \leq \text{sizeof(int)} \leq \text{sizeof(long)} \leq \text{sizeof(float)} \leq \text{sizeof(double)}$

- Operatii Arithmetice:

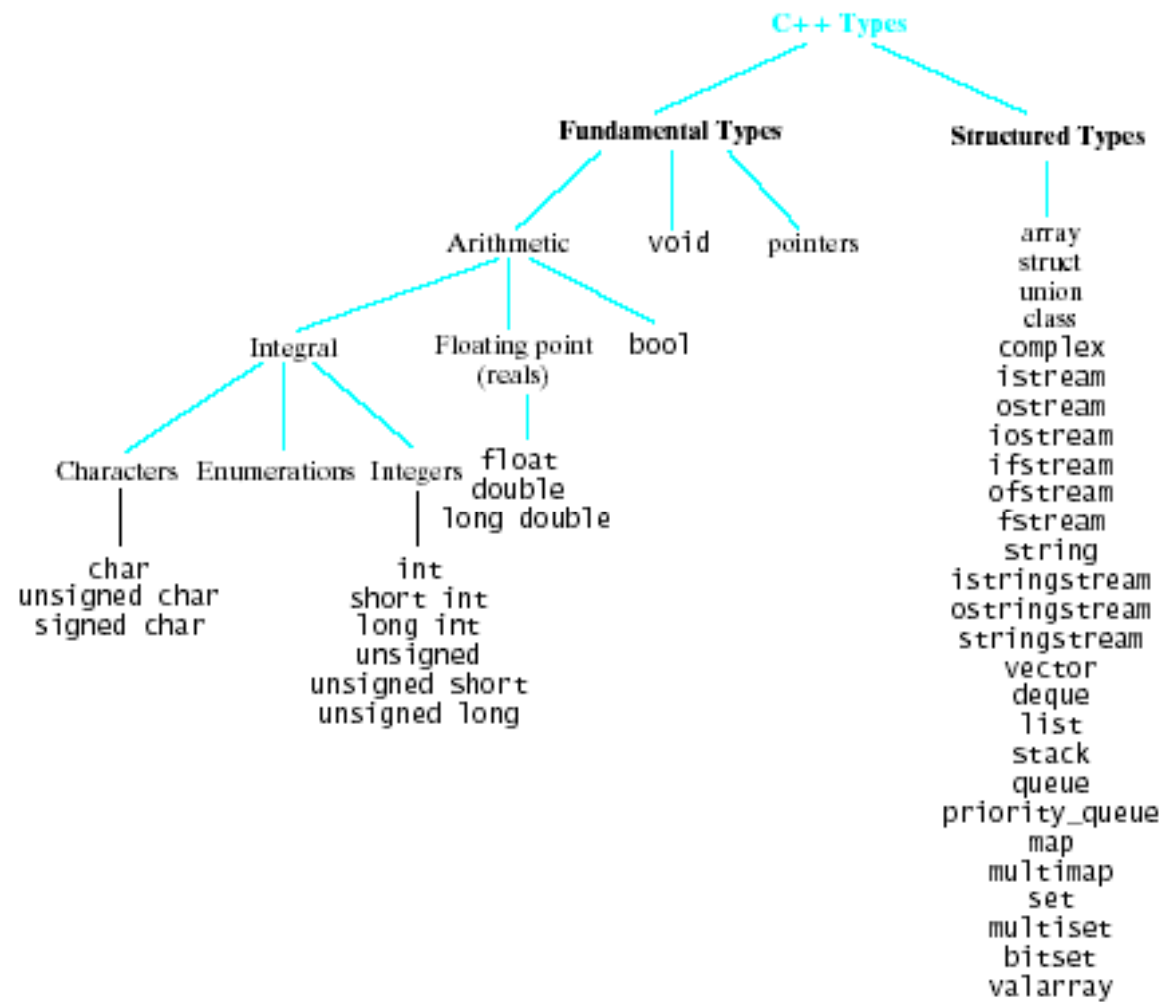
$+$, $-$, $*$, $/$, $\%$ (modulo)

Ex. $\text{int } i = 3, j = 2; \quad i / j = 1; \quad //$ integer division

- Relatii:

$<$, $>$, \leq , \geq , $=$, \neq

Tipuri C++



Domeniu de vizibilitate

domeniu de vizibilitate a unui nume(identificator)
= fragmentul de cod sursa unde acesta poate fi folosit

Vizibilitate de tip bloc:

- blocul este definit de acolade: { }

Exemplu: program C++ program format dintr-un fisier sursa

```
int x =1;      // visibility from this point to the end of file
{ int a;
    int x =2;  // the global variable is hidden by the local variable
    ::x =3;    //the global variable
} // a is not visible any more
int y;
// x has the value 3
```

In acelasi bloc este interzisa declararea mai multor variabile cu acelasi nume!

Vizibilitate

- **Module**

- *Example:*

`iostream.h` - contains input/output declarations

Un program C++ program care consta din:

- fisierul `iostream.h` (`cin`, `cout`)
- fisierul – cod sursa

`p1.cpp` contine:

```
#include <iostream.h>
```

```
int a, b;
```

```
long c;
```

```
....
```

`cin`, `cout`, `a`, `b`, `c` sunt vizibile de la punctul lor de definire – tot programul (vizibilitate globala).

- vizibilitate globala intr-un fisier doar:

```
static int a;
```

- folosirea unei variabile definita in alt fisier:

```
extern double x;
```

Operatorul de rezoluție (::)

- **Operatorul de rezoluție (::)** se utilizează când dorim să referim o variabilă globală redefinită într-o funcție:

::<variabila>

Exemplu:

```
#include <conio.h>;
#include <iostream.h>;
unsigned x; // Variabile Globale (::)
char y[10];
int i = 99;
void main (void)
{ clrscr();
  unsigned x; // Variabile Locale
  int y;
  cout << " Dati doua numere : "; cin >> ::x >> x;
  cout << " Cele doua numere = " << ::x << ' ' << x << endl;
  cout << " Cele doua numere = " << ++::x << ' ' << --x << endl;
  cout << " Cele doua numere = " << ::x << ' ' << x << endl;
  cout << " Dati nume,varsta : ";
  cin >> ::y >> y;
  cout << " Numele si varsta = " << ::y << ' ' << y << endl;
  for (int i=11; i<::i; ::i=11) cout << i << " " << ::i << endl;
  getch();
}
```

Operators

- arithmetic
 - ++ (unary) increase argument by one
 - (unary) decrease argument by one
 - +, -, *, /, % (binary)
- bits op.
 - &, |, ^, <<, >> (binary) and, or, xor, shift left, shift right
 - ~ (unary) not
- relational
 - ==, !=, <, >, <=, >= (bin) equal, not equal, less, ...
- logical
 - ! (unary) boolean not
 - &&, || (binary) boolean and, boolean or
- assignment
 - = (binary)
 - arith. assign +=, -=, *=, /=, %= (bin) add to, subtract from
 - bit assign. &=, |=, ^=, <<=, >>=

Precedenta operatorilor

$a+b*c \rightarrow a+(b*c)$

:

++, --, ~, !, *, /, %, +, -, <<, >>, <, <=, >, >=, ==, =

Remarci:

- orice expresie C++ are un rezultat numeric
- nu exista valori booleene:
orice $\neq 0$ is **true**, si 0 reprezinta **false**
- nu exista expresii logice

Operatia de atribuire

- **Sintaxa:**

Var = Expresie; // expresie de atribuire cu tipul Var

- Atribuirii multiple prin expresii de forma:

Var1 = ... = Var2 = Var1 = Expresie;

- operator + Atribuire

Var □ = Expresie; // unde □ ∈ {+, −, *, /, %, &, ^, |, <<, >>}

- având semnificația:

Var = Var □ Expresie;

Operatorii de incrementare / decrementare

- **Operatorii de incrementare / decrementare** sunt '++' respectiv '--' prin care se mărește, respectiv se micșorează, valoarea operandului cu unu.
- Aceștia pot fi utilizați:
 - în forma prefixată:
 - ++ operand ; respectiv -- operand ; // valoarea expresiei, după aplicarea lor
 - în forma postfixată:
- operand ++; respectiv operand --; // valoarea expresiei, înainte de aplicare

Operatorii condiționali ?

- **Sintaxa:**

Expresie1 ? Expresie2 : Expresie3 ;

- Valoarea expresiei rezultat este Expresie2 dacă Expresie1 este nenulă, altfel este Expresie3 .

- **Exemplu:**

Max = a>b ? a : b; // Dacă a>b Atunci Max=a Altfel Max=b;

Operatorul de conversie explicită

- **Operatorul de conversie explicită** (expresie cast) realizează conversia unui operand într-un tip precizat astfel:

(Tip) operand ;

- Exemplu:

```
int a=12; int b=5;
```

```
float c=a/b; printf(" a Div b = %5.2f \n", c); // a Div b = 2.00
```

```
c=(float)a/b; printf(" a / b = %5.2f \n", c); // a / b = 2.40 05.03.06 10
```

- **Conversiile implicite** (realizate automat) :

a) char→int,

b) float→double,

Operații pe biți

- **Operații pe biți :**
- \sim (complementul față de FFFF, schimbă fiecare bit),
- \ll (deplasare la stânga),
- \gg (deplasare la dreapta),
- $\&$ (And bit cu bit),
- \wedge (Xor bit cu bit),
- $|$ (Or bit cu bit).

Instrucțiuni

■ Instrucțiunea Vidă

- Această instrucțiune se utilizează în situația în care este nescerară prezența unei instrucțiuni și care nu trebuie să execute nimic:

;

■ Instrucțiunea compusa { <instr1>; <instr2>; }

■ Instrucțiunea If

if (expresie) instructiune1;
[else instructiune2;]

■ Instrucțiunea Switch

- structura alternativă cu mai multe ramuri Case (Select) :

switch (<expresie>)

{ case c1 : <secvență instructiuni 1> [break;]

case c2 : <secvență instructiuni 2> [break;]

...

case cn : <secvență instructiuni n> [break;]

[default : <secvență instructiuni n+1>]

}

- Instrucțiunea **break** realizează saltul la sfârșitul instrucțiunii **switch**, iar în absența ei se vor executa și următoarele secvențe de instrucțiuni.

Instrucțiuni de ciclare

■ Instrucțiunea While

- Structura repetitivă pretestată :

while (<expresie>) <instrucțiune>;

■ Instrucțiunea Do_While

- Structura repetitivă posttestată poate fi scrisă astfel:

do <instrucțiune> **while** (<expresie>;

■ Instrucțiunea for

- Numar predefinit de pasi

for (<init>; <cond>; <avans>) <instrucțiune>

Referinte

- A.V. Aho, J.E. Hopcroft, J.D. Ullman, Data Structures and Algorithms, Addison-Wesley Publ., Massachusetts, 1983.
- R. Andonie, I. Garbacea, Algoritmi fundamentali. O perspectiva C++, Editura Libris,
- Alexandrescu, Programarea moderna in C++. Programare generica si modele de proiectare aplicate, Editura Teora, 2002
- M. Frentiu, B. Parv, Elaborarea programelor. Metode si tehnici moderne, Ed. Promedia, Cluj-Napoca, 1994.
- E. Horowitz, S. Sahni, D. Mehta, Fundamentals of Data Structures in C++, Computer Science Press, Oxford, 1995.
- K.A. Lambert, D.W. Nance, T.L. Naps, Introduction to Computer Science with C++, West Publishing Co., New-York, 1996.
- L. Negrescu, Limbajul C++, Ed. Albastra, Cluj-Napoca 1996.
- Dan Roman, Ingineria programarii obiectuale, Editura Albastra, Cluj_Napoca, 1996.
- B. Stroustup, The C++ Programming Language, Addison Wesley, 1998.
- Bruce Eckel, Thinking in C++, www.bruceeckel.com